Big Picture:
Graphs can be used to model a lot of real-life situations, including computer networks. Large computer networks contain many nodes and can become disconnected if some nodes within the network fail. In this assignment you will use a graph to represent a simple computer network, and you will use some graph algorithms to determine the current status of the network. As nodes fail you will remove them from the network, and as they are repaired you will return them to the network. Edges may also change as the quality of various connections changes over time. As the network changes you will report its status, specifically each of the following:
1) Whether or not the network is currently connected
2) Which nodes are currently up or down in the network
3) A listing of the current subnetworks and their vertices and edges (i.e. the connected components within the network)
4) A minimum spanning tree (by latencies) for the network
5) The shortest (i.e. smallest latency) path from arbitrary vertex i to arbitrary vertex j.
6) The distinct paths from vertex i to vertex j whose latencies sum to a value <= x.

Details:
You must write a single class, called Graph, which will implement the functionality described above and below. Initially you will read in a graph from a file, formatted in the way indicated in your textbook. The first line will contain an integer representing V, the number of vertices in the graph. The second line will contain an integer representing E, the number of edges in the graph. The remaining E lines will contain triples to represent the actual edges in the graph. The first two values in the triple will be the two vertices that terminate the edge and the third value will be the weight of the edge, which in this case will be the latency for the edge (so a smaller weight indicates a faster traversal on the edge). All edges will be bidirectional, so if edge (a, b) has weight w, then edge (b, a) also has weight w. For example, the simple graph shown below will be represented by the file shown in the box immediately to the right of it.

Updates to the graph and the report options will be obtained via interactive user input. Below are a list of the commands and the expected results:
- **R (eport)** – display the current active network (all active nodes and edges, including edge weights);
  - show the status of the network (connected or not);
  - show the connected components of the network
• **M** (minimum spanning tree) – show the vertices and edges (with weights) in the current minimum spanning tree of the network. If the graph is not connected this command should show the minimum spanning tree of each connected subgraph.

• **S i j** – display the shortest path (by latency) from vertex i to vertex j in the graph. If vertices i and j are not connected this fact should be stated.

• **P i j x** – display each of the distinct paths (differing by at least one edge) from vertex i to vertex j with total weight less than or equal to x. All of the vertices and edges in each path should be shown and the total number of distinct paths should be shown at the end.

• **D i** – node i in the graph will go down. All edges incident upon i will be removed from the graph as well.

• **U i** – node i in the graph will come back up. All edges previously incident upon i (before it went down) will now come back online with their previous weight values.

• **C i j x** – change the weight of edge (i, j) (and of edge (j, i)) in the graph to value x. If x is <= 0 the edge(s) should be removed from the graph. If x > 0 and edge (i, j) previously did not exist, create it.

• **Q** – quit the program

**Implementation:**
You are required write a single class called Graph which enables the functionality described above. Your main program should include a menu-driven loop which utilizes the appropriate methods in your Graph class to execute each of the commands. Your class should use an adjacency list to represent your graph. You may co-opt / utilize any of the author’s code that you wish, but you must a) credit the author and b) modify it as necessary to fit within your class framework. Here are a few important implementation requirements and issues that are worth noting:

- The graph file name should be entered as a command line argument to your main program
- The author's implementations do not take into account removal of vertices or edges. You will need to adapt the implementations to handle this ability. There are various ways of doing this but note that when you restore a vertex its edges are also restored, so you need to store the edge information even when a vertex is down.
- The author uses different classes for the minimum spanning tree (EdgeWeightedGraph) and the shortest path tree (EdgeWeightedDigraph). As we discussed in lecture, we can easily make a directed graph with bidirectional links by simply adding an edge in each direction (a, b) and (b, a) for each bidirectional edge (a, b). Take this into account when building your graph, since the file format includes only one vertex pair for each edge.
- The shortest path code provided by the author gives the shortest path from the source to each of the other vertices in the graph. Clearly, you should not display all of these, so you should modify this code.
- The P i j x command is non-trivial and will require a recursive backtracking algorithm. As a hint think about how permutations of the vertices would be generated, and how that algorithm would be modified to take into account actual graph edges and the bound on the path weight.
- See the CS 1501 Web site for some test files and some example output.

**W Section:**
Write a short (~5 pages, double spaced, 11-point) biography on Donald Knuth. Include some of his accomplishments as well as any other information that you would consider to be interesting or worth noting. You must cite two valid non-Wikipedia references in your paper. In this case valid means a source that is officially published – either a paper or book.

**Extra Credit:**
There are a lot of interesting extra credit options possible for this assignment. Below are a few possibilities:

- Allow new vertices to be added to the network. If you allow this option you should also prompt the user for all of the edges associated with a new vertex.
- Show the graph or tree in a graphical way
- In addition to the normal P i j x command, implement a version where paths have no common edges

**Submission:**
Zip all of your source and data files (**including any of the author's files that you use**) together and submit a single .zip file to the submission site by the appropriate deadline. As usual, your program should compile and run using only the files that you submit.